5 Months of Haskell : Programming languages as just other programs. Contribution toward a field of *computer science education.*

Camille Akmut

Abstract

The following is understood as a contribution toward a field of *computer science education* : a reflection of 5 months of learning the functional programming language Haskell; out of which has emerged for us that programming languages are 'just other programs'. This lesson, so important, is never felt more than in a functional language like Haskell, we defend. It has for principal benefit to bring down the barriers between creators and users of programming languages, i.e. "programmers", both are the same; a psychological-sociological fact not without revolutionary characteristics.

Introduction : on riding bikes, and violins

There *is* something magical about computer science, and programming, and programming languages – we must leave that to Abelson and Sussman. But, not in the commonly understood way, that was not theirs.

Learning a programming language, though no different perhaps than any other task, is very much like learning to ride a bike, or learning how to swim : this, in fact, *strange, strange* process by which a skill is learned, but the process forgotten!

This has countless implications and complications, of course. On students, and on their teachers.

The psychology and sociology of these events, as they specifically relate to programming, escapes us, yet. It is doubtful whether the already established field of "mathematics education" – which may or may not serve as a model – has solved its own problems[1].

—

The great computer scientist Edgar Dijkstra, perhaps no other had blurred the lines between computer and social sciences (they are the same, we repeat, and maintain) more than him, wrote :

> *It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use, and in this respect programming languages [are no different]: they shape our thinking habits.*[2]

Out of which follows, that computer scientists, and whatever administrators have greater say than them, carry a huge responsibility when picking a programming language for their students. – so Dijkstra, so us.

—

And, so, to avoid this, which is to say the loss of memories attached to a learning process, as it had been for myself, I write them down in the hopes they will be useful to future researchers of this field.

[1] See, for instance, Alcock 2013, whose approach is not entirely convincing.
[2] Dijkstra 2001.

1 Background

Little would be learned from my experiences, if I did not provide some indications about my background. In writing this text, I must also alternate constantly, and awkwardly, between the "I" of the participant, the learner, and the "we" of the observer, scientist.

—

Contrary to many others, it appears to me, I say this as some sort of feeling, but not as a matter of fact[3], I had learned programming as an adult : I was well into my twenties, late twenties in fact!

—

It is important to publicize and communicate such experiences, not only to break with the myth of the "*genius*" programmer (least the even more improbable notions of "rock star programmers", notions so laughable, they can only be the product of an even more so industry); but

"Genius", with a capital, only, because, like virtuoso violinists, they had started early on; the only fact more improbable would be for anyone to not reach some level of mastery after 10 or 20 years of practice; but, these fools, would prefer to cover and erase the steps and tracks and acts of their painful and long learning journeys, rather than admit to it; and, encourage others.

—

My "first language", by which I specifically mean the first language that I had discovered for myself as a "proper language", this is to mean one that I would enjoy writing, thinking, and expressing myself in, was Python. As has already been noted many times, its appeal, so for myself also, was that it looked like English. There was something clean about it, all unnecessaries removed. (My first language was Wirth's Pascal[4].)

—

Before starting with Haskell, I had been programming in Python for about 2 years. At which point, I had become able to "translate" small real-world problems into code i.e. performing security/network tasks on many servers at once, automatically.

—

Languages that I simply cannot bring myself to think in, or write in, or that I generally want as little to do with as possible, include :

– Java/C# : heavy, heavy dislike. – On an almost instinctive level, my feeling; and later opinion. I understood just enough of what it was trying to do, to also understand I wanted nothing to do with it. This verbose, corporate language... Half the time, and energy is spent instructing this language's compiler, rather than expressing ideas. A 12-line "Hello World" program should have convinced anyone that perhaps something had gone wrong during its design. "*What a* horrible *language*", thought Torvalds out loud, and Dijkstra much worse. Used for decades as first language... (The 1990's "workhorse".)

[3] Studies would be needed.

[4] Learned in a high-school programming class, soon dropped, not because of the language, which is fine. (Wirth's ideas, e.g. contra-OOP, are compatible with functional programming?)

– JavaScript : a "week-end affair", created by a homophobe with remorses the night-after, what a wonderful background story for any language to have... Half the time, and energy, here, is spent correcting the mistakes of the language, *that are the language itself.*

Languages that I have enjoyed in the past, or don't mind using :

– Rust. The best compiler I have ever interacted with. Programming, here, takes on the traits of a dialog between a student and teacher, with the former informing of errors, their nature, *and how to correct them* – precisely, correctly, and accurately. (*In theory*, all compilers.)
– "Lisp-family programming languages". I liked Scheme/Racket, Clojure. I feel most at home now with functional languages.
– Ruby. Although I agree with the sentiment of one StackOverflow user, that "*Ruby is Candy Coated Perl, just as Macs are Candy Coated UNIX, the two are attached respectively to each other.*"; it is not a bad language. Inspired by Python, resembling it, but making it also more clunky in some respects (begin/end...).

2 Beginnings

The earliest "proof" of my involvement with Haskell, that I could find, comes in the form of a commit with beginning SHA `19f3ddd1` for a program `quicksort.hs`, dated "01 Jan, 2019".

This is also how I remember it : first of January. I had finally managed to make the Haskell compiler work for me, after much struggles to bring it to do, or validate *anything.* And, with this victory, I had resolved to only program in Haskell for an entire year – so, in theory, anyway...

Listing 1: quicksort.hs

```
-- Graham Hutton, Programming in Haskell.


qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
    where
        smaller = [a | a <- xs, a <= x]
        larger = [b | b <- xs, b > x]


main = print(qsort [1,5,6,2,13,2,3,2])
-- output: [1,2,2,2,3,5,6,13]
```

I think, this was the moment "I fell in love" with Haskell, and functional programming in general perhaps. But, specifically Haskell. – and, because we lack yet the vocabulary to express these happenings correctly, what these analogies lack in precision, they make up with immediacy.

At that point, and in my mind, the principal "frame of reference" I had of this algorithm – the ways in which it could be expressed, and specifically the most simple, clean ones – was the implementation of the mergesort algorithm that can be found in John Guttag's *Introduction to Computation and Programming Using Python* (they had left out quicksort, because it was even more complicated they said!) :

Listing 2: recursive-mergesort.py

```
"""As presented by Eric Grimson, based on John Guttag's book,
with additional comments and modifications by me"""


def merge(left, right):
    result = []
    i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while i < len(left):
        result.append(left[i])
        i += 1
    while j < len(right):
        result.append(right[j])
        j += 1
    return result


def merge_sort(L):
    if len(L) < 2:
        return L
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)


array1 = [2, 5, 4, 123, -4, 0, 2, 1]
array2 = []
array3 = [-1]
print(merge_sort(array1), merge_sort(array2), merge_sort(array3))
# [-4, 0, 1, 2, 2, 4, 5, 123] [] [-1]
```

Needless to say, my mind was made up.

I had not been convinced through arguments, or sound theories : but, by a matter-of-fact, very real and almost palpable difference, obtained trough the most summary of comparison.

—

On my computer, I have another file, `p.hs`, dated "Mo 31 Dec 2018", created shortly before midnight...

Listing 3: p.hs

```
sum2 [] = 0
sum2 (n:ns) = n + sum2 ns
main = print(sum2 [1,2,3])
```

—

These were my beginnings in Haskell, best I can recall.

3 A lesson learned early : programming languages are just 'other programs'

I have used, continue to use primarily two books to teach myself Haskell, these are :

1. Graham Hutton's *Programming in Haskell*; and
2. Simon Thompson's *The Craft of Functional Programming.*

Additionally, also :

3. Miran Lipovaca's *Learn You a Haskell for Great Good!*

And, 4., the variety of Internet and Web ressources that are available[5].

—

Common to all of them, 1. certainly, 2. in older editions[6], and perhaps also 3., is the lesson whose ending we will spoil here : programming languages are just 'other programs'. What do we mean by this?

All of the aforementioned textbooks have for common trait that from the very first chapters, if not pages on parts of the programming language Haskell itself are reprogrammed by the reader, learner.

Such that, by the end, significant portions of Haskell – as can be found in Prelude – have been reprogrammed, all while learning to program.

This occurs, for instance, in 6.2 Polymorphism of an earlier edition of Thompson's book, where the length function is found

Listing 4: length

```
length [] = 0
length(a:x) = 1 + length x
```

The same is found literally everywhere in Hutton's book (of which, we have read the entire first part, corresponding to an undergraduate course.)

—

[5] https://www.haskell.org/community/

[6] We must be a little harsh here, in saying that we did not care *at all* for the "horse" example of the latest edition, the third. Was this a pedagogical phase of the time?

This is, simply, not the way this is approached in most other languages, and their textbooks : it would not occur to any beginning Python learner to re-program Python, and it would not because this part of intellectual activity is not emphasized, and it is not because in these languages doing so would be either fastidious or only reserved to advanced programmers.

The *summum*, so to speak, of a learning journey – only accessible to wizards.

—

But, not so in Haskell, or other functional programming languages, and their textbooks[7] : here it is neither the beginning, not the end.

The end, or beginning perhaps we should say, left implicit, though nonetheless real, seems for teachers like Hutton and Thompson to bring their readers and students to a point where they would be able to program, if not new languages, then at the very least so-called "domain specific languages" (written in Haskell).

—

Neither the beginning, nor the end – as we have said.[8]

[7] A pedagogical innovation started by Abelson and Sussman in *Structures...*?

[8] During my journey so far, I have submitted many errata, and it made me feel, rightly or wrongly, that I had become part of a community. – This, I cannot say about any other language(s).

References

—. 2019. “What is Computer Science? Outline for a project.”

—. 2019. “Computer science is a social science.”

Abelson, Harold and Sussman, Gerald. *. *Structure and Interpretation of Computer Programs.*

Alcock, Lara. 2013. *How to Study for a Mathematics Degree.* Oxford University Press.

Dijkstra, Edgar. 2001. “To the members of the Budget Council”. https://www.cs.utexas.edu/users/EWD/transcriptions/OtherDocs/Haskell.html

Hutton, Graham. 2016. *Programming in Haskell.* Cambridge University Press. Second edition.

Lipovaca, Miran. *Learn You a Haskell for Great Good!* http://learnyouahaskell.com/

Thompson, Simon. 2011. *Haskell: the craft of functional programming.* Addison-Wesley. Third edition.